PERBANDINGAN ANTARA C++ DAN JAVA DALAM PENGELOLAAN SUMBER

Subandijo

Computer Science Department. School of Computer Science Binus University
Jl. K.H. Syahdan No. 9 Palmerah Jakarta Barat 11480
subandijo1030@gmail.com

ABSTRACT

This article contains a descriptive comparison of four features between C++ and Java. The first feature discusses about memory management that focuses on exception safety. The second compares the C++ destructor to Java finalizer. Third feature discusses about Java finally block. The forth is about C++ and Java implementation in order to access a data structure.

***Keywords**: exception safety, destructor C++, Java finalizer, Java finallyblock*

ABSTRAK

Artikel ini berisikan uraian deskriptif perbandingan empat fitur antara bahasa C++ dan Java. Fitur pertama mendiskusikan pengelolaan memori yang difokuskan pada keamanan eksepsi, yang kedua membandingkan destruktor C++ dan Java finalizer, ketiga menguraikan Java finally block dan terakhir implementasi C++ dan Java untuk mengakses struktur data.

***Kata kunci:** keamanan eksepsi, destruktor C++, Java finalizer, Java finally block*

PENDAHULUAN

Studi perbandingan antara bahasa pemrograman C++ dan Java telah banyak dikaji tetapi umumnya studi ini dipelajari berdasarkan asal usul pengembangan dan tujuannya saat mereka dikembangkan. C++ yang dirancang sebagai bahasa pemrograman aplikasi dan pemrograman sistem merupakan ekstensi dari bahasa C. Kepada bahasa C yang berupa bahasa prosedural C++ menambahkan sejumlah fitur POO (pemrograman orientasi obyek), yaitu TDA (tipe data abstrak), inheritance dan pengikatan dinamik untuk mengimplementasikan *true polymorphism*. Selain itu juga ditambahkan pemrograman generik dan STL (*standard template library*) yang mencakup kontainer generic dan algoritma.

Di sisi lain, Java awalnya dikenal sebagai bahasa interpreter untuk sistem pencetakan tetapi kemudian berkembang untuk mendukung pemrograman jaringan. Java adalah bahasa OO yang menggunakan sintaks mirip C++ tetapi tidak kompatibel dengannya. Ia dirancang untuk mudah digunakan dan dapat diakses oleh banyak pengguna. Karena itu membandingkan keduanya adalah isu yang sangat menarik, semata-mata bukan karena kepopuleran mereka sebagai bahasa OO tetapi karena hingga sekarang kedua bahasa tersebut terus menerus dikembangkan oleh masing-masing *developer*nya.

METODE

Studi deskriptif ini merupakan studi literatur. Studi tentang perbandingan kedua bahasa telah banyak diulas oleh banyak peneliti tetapi umumnya mereka hanya mencakup perbedaan sintaks di antara kedua bahasa yang dibahas secara *head to head*. Karena itu studi ini tidak membandingkan semua fitur yang dipunyai oleh kedua bahasa tetapi hanya membandingkan sejumlah komponen yang dirasa sangat kontras di antara keduanya dan tentu saja jarang atau tidak banyak diulas oleh studi-studi sebelumnya. Komponen-komponen yang dimaksud adalah pengelolaan memori, destruktor C++ vs *finalizer* Java, blok *finally* Java dan akses kedua bahasa ke struktur data yang sama. Selain itu sejumlah penggalan kode juga disajikan untuk memperdalam analisis yang dilakukan.

HASIL DAN PEMBAHASAN

Pengelolaan Memori Dinamik

Kedua bahasa menggunakan konsep yang jauh berbeda. Tidak seperti Java yang menawarkan *garbage collection* (GC) secara otomatis, pengelolaan memori di C++ biasanya dilakukan menggunakan konstruktor, destruktor dan *smart pointer*. Meskipun C++ baku mengijinkan penggunaan GC melalui kepustakaan standar *libgc* (Kapoor, 2003) tetapi secara praktis GC jarang digunakan. Alasan utamanya adalah sulitnya menulis *real time software* bagi pemrogram C++ jika menginginkan penggunaan GC dilakukan secara otomatis. Dengan demikian C++ secara *default* tidak menggunakan GC.

Obyek C++ secara fisik diletakkan di tiga lokasi berbeda, tergantung pada tipenya, yaitu di memori statik, di *stack* untuk obyek lokal dan di memori *heap* untuk obyek dinamik. Obyek di memori statik adalah obyek-obyek yang di bentuk saat program mulai dieksekusi dan akan selalu tersedia sampai program berakhir. Peubah global dan *namespace* selalu diletakkan di memori statik. Hal yang sama berlaku juga untuk peubah fungsi dan klas yang menggunakan *modifier static*. Peubah statik

suatu fungsi tidak dibentuk saat program mulai dijalankan tetapi deklarasi mereka pertama kali dikenali selama program dieksekusi. Obyek statik dikonstruksi hanya sekali yaitu saat program dieksekusi dan dihapus saat program selesai dieksekusi. Obyek statik tidak didestruksi jika program berakhir karena pemanggilan fungsi *exit(), abort()* atau fungsi-fungsi serupa. Kebutuhan memori ditangani oleh kompilator.

Obyek yang disimpan di *stack* sepenuhnya ditangani oleh kompilator. Tidak ada kebutuhan sama sekali untuk menghapus mereka sepenuhnya dari memori meskipun mereka sudah tidak dibutuhkan. Meskipun demikian ada beberapa batasan agar kompilator bisa mengelolanya secara efisien. Hanya obyek yang ukurannya statik yaitu obyek yang ukurannya diketahui saat kompilasi dan obyek-obyek yang *lifetime* nya tidak melampaui blok di mana mereka dideklarasikan yang dapat disimpan di *stack*. Ada catatan kecil yang perlu diperhatikan. Faktanya setiap obyek mempunyai ukuran statik saat dikompail yang ukuran riilnya dapat diketahui dari operator *sizeof*. Artinya setiap obyek dapat diletakkan di *stack*. Tetapi ini tidak berarti bahwa semua persyaratan memori untuk obyek ini dapat dipenuhi oleh *stack*. Jika kita ingin meletakkan *string* di *stack*, implementasi dari *string* akan membutuhkan memori dari *heap* untuk memenuhi tugas ini. Penggalan kode berikut menunjukkan bagaimana dua obyek yaitu *x* dan *y* disimpan di *stack*.

```
// Penggalan kode 1
01 void stackAllocation() {
02  int x = 0;
03  {
04    x = 3;
05    int y = x;
06    ++y;
07  }
08  x++;
09 }
```

Peubah x dideklarasikan di baris 2 dan tetap ada di *stack* sampai blok di mana ia dideklarasikan berakhir. Di antara deklarasi sampai blok berakhir peubah x dapat digunakan. Sebaliknya peubah y dikonstruksi dibaris 5 dan dihapus dibaris 7. Dapat dilihat konstruksi obyek dilakukan secara eksplisit tetapi destruksinya dilakukan secara otomatis oleh kompilator. Kompilator akan memanggil destruktor untuk obyek yang bersangkutan dengan urutan yang berlawanan saat konstruksi. Catatan penting adalah obyek di *stack* umumnya tidak mungkin *leak.*

Obyek di C++ juga mungkin di simpan di *heap*. Obyek tipe ini dialokasikan secara dinamik saat eksekusi. C++ menggunakan operator *new* untuk mengalokasikan memori dinamik. Obyek yang dibentuk menggunakan operator *new* tetap ada di memori sampai ia dihapus menggunakan operator *delete*. Operator *new* dan *delete* beroperasi pada tipe data *pointer*. Kode berikut menunjukkan bagaimana dua tipe data integer dialokasikan di *heap*.

```
// Penggalan kode 2
01 void heapAllocation() {
02  int* x = new int (0);
03  int z;
04  {
05    *x = 3;
06    int& y = new int (*x);
07    y++;
08    z = &y:
09  }
10  (*x)++;
11  delete x;
```

```
12  ++(*z);
13  delete z;
14 }
```

Obyek pertama di konstruksi di baris 02 dan disimpan di peubah x. Obyek ini eksis sampai di hapus di baris 09. Cara lain membentuk obyek dinamik yaitu menggunakan operator *reference* seperti yang terlihat di baris 05. Dalam hal ini y menjadi alias atau nama lain dari obyek yang dibentuk di *heap*. Selama belum dihapus peubah y dapat digunakan seperti halnya kita menggunakan peubah yang ada di *stack*. Peubah y tidak dapat diakes di dalam blok dalam karena lingkup dia adalah hanya sampai baris 09. Peubah y akan dihapus saat eksekusi mencapai baris 09. Untuk mengakses di luar blok maka ia kita salin ke peubah z yang tetap eksis di luar blok. Catatan bahwa *pointer* dan *reference* untuk mengakses obyek di *heap* dialokasikan di *stack*.

Meskipun teknik ini sangat luwes tetapi ada biaya yang harus dibayar. Apa yang kita lakukan harus kita kelola sendiri. Kompilator tidak akan menghapus obyek untuk kita dan membolehkann kita untuk menggunakan pointer ke obyek yang telah dihapus. Tetapi masalah utamanya adalah kita harus mengelola pembersihan memori yang dialokasikan secara dinamik termasuk jika ada eksepsi seperti yang terjadi di baris 06 karena obyek di baris 02 akan hilang sehingga memori yang ditempatinya akan *leak*. Untuk mengatasi masalah ini C++ menggunakan tiga pendekatan, yaitu: (1) jika mungkin gunakan pendekatan GC; (2) hindari pengelolaan memori dinamik secara manual; (3) gunakan *smart pointer*.

Meskipun tidak popular, C++ mempunyai subsistem yang mengelola GC mirip dengan Java. Perlu dicatat bahwa banyak kolektor GC mempunyai sejumlah limitasi: mereka tidak bisa menghapus obyek yang telah dikoleksi. Ini berarti bahwa destrukor obyek tersebut tidak akan dipanggil. Dengan kata lain kolektor GC tidak menghapus obyek secara benar, mereka hanya menggunakan kembali memori yang ditempati oleh obyek tersebut. Deskripsi lain dari teknik ini adalah model "*infinite memory*" yang berarti hanya sumber yang tepat yang perlu dikumpulkan.

Pendekatan kedua dilakukan dengan menggunakan STL yang menyediakan banyak implementasi klas generik. STL adalah bagian dari spesifikasi C++ sehingga tersedia di semua *platform* C++. Mayoritas klas STL menyediakan beberapa kontainer seperti *link-list*, *hash table* dan *string*. STL dapat menangani memori dinamik untuk kita sehingga mereduksi keinginan untuk menanganinya secara manual.

Java mempunyai sistem GC yang *built in*. Berdasarkan pada keterjangkauan obyek, JVM (Java Virtual Machine) mulai proses finalisasi. Gosling, Joy, Steele dan Bracha (2000) menyajikan diagram *state* komplit yang mencakup semua delapan obyek *state* yang mungkin. Dua kunci utama obyek adalah keterjangkauan obyek dan finalisasi. Keduanya mempunyai tiga nilai yang mungkin. Obyek mungkin *reachable*, *unreachable* dan *finalized*. Kedua *state* membentuk matriks 3 x 3 yang mencakup semua nilai obyek *state* yang mungkin di mana satu kondisi (*unreachable* dan *finalized*) tidak mungkin terjadi. Model rumit ini dibutuhkan untuk membuat referensi sirkular GC mungkin dilakukan. Hal ini merupakan fitur penting untuk membebaskan terjadinya *leak* GC untuk proses eksekusi yang panjang di mana sirkular referensi kerap muncul seperti obyek A merujuk obyek B dan sebaliknya. Penghitungan murni referensi didasarkan pada GC tidak mampu mengkoleksi referensi sirkular, banyak diantaranya membutuhkan pengembang untuk *break* dari *loop*.

Destruktor vs Finalizer

C++ mempunyai destruktor sedangkan Java mempunyai *finalizer*. Keduanya mempunyai fungsi yang sama dan dipanggil pada saat yang sama sebelum dealokasi obyek tetapi mereka berbeda signifikan dalam sejumlah hal. Kedua bahasa membolehkan *cleanup* (pembersihan) data yang

dieksekusi sebelum *life time* obyek berakhir. Satu-satunya perbedaan adalah obyek *undefined* jika Java *finalizer* dipanggil sedangkan di C++ obyek sepenuhnya *defined* jika destruktor dipanggil. Metode pembersihan data biasanya dibutuhkan untuk membebaskan sejumlah sumber yang terikat di konstruktor. Jika kita mengalokasikan memori dinamik di konstruktor obyek C++ kita harus mendealokasi memori ini jika sudah tidak dibutuhkan. Penggalan kode berikut menunjukkan bagaimana destruktor beroperasi di klas C++.

```
// Penggalan kode 3
01 class cppDealokasi {
02   public:
03     cppDealokasi(): _p (new int())
04     {
05     };
06     ~cppDealokasi()
07     {
08        delete p;
09     }
10   private:
11     int _p;
12 }
```

Konstruktor melakukan inisialisasi peubah anggota *private* _p dengan *pointer* peubah dinamik tipe integer di baris 3. Obyek dapat menggunakan integer ini selama *life time* nya tetapi jika *cppDealokasi* menghapus obyek maka destruktor akan dipanggil untuk menghapus obyek yang ditunjuk oleh p (baris 8).

Contoh yang mirip ini di Java tidak memerlukan *finalizer* karena *garbage collector* berkepentingan dengan kasus ini. Dengan alasan ini *finalizer* Java jarang digunakan dibandingkan dengan destruktor C++. Kekurangan solusi Java adalah ia bekerja sangat baik untuk memori tapi kurang memperhatikan sumber lain seperti *file handle*, *open socket* dan yang mirip dengannya. Dengan C++ kita cukup menggunakan teknik yang sama, tetapi dengan Java penggunaan *finalizer* tidak cukup karena *undefined* saat ia dipanggil. Dengan alasan ini Java *finalizer* sangat tidak *reliable*. Ini merupakan alasan lain mengapa mereka tidak digunakan. Jika kita sepenuhnya memfungsikan destruktor di Java satu-satunya jalan adalah dengan mengimplementasikan sebagai metode publik yang melakukan pembersihan dan dipanggil secara eksplisit.

Blok Finally

Kedua bahasa mengenal eksepsi dan mendukung blok *try-catch* untuk menganganinya. Tetapi hanya Java yang menggunakan blok f*inally*. Blok ini mengikuti blok *try* dan dapat memuat kode yang akan dieksekusi meskipun blok *try* telah selesai dieksekusi. Ini berarti bahwa kode di blok *finally* akan dieksekusi jika blok *try* dieksekusi secara normal dalam arti tidak ada eksepsi seperti halnya jika ada eksepsi dan mungkin jika ada dua, satu di blok *try* an satu di blok *catch*. Blok *finally* selalu dieksekusi sebagai bagian terakhir dari blok *try* sehingga jika ada blok *catch* maka blok *catch* diekekusi pertama kali jika ada eksepsi. Dengan alasan ini blok *finally* kerap digunakan untuk melakukan tugas pembersihan. Contoh dari Java finally adalah sebagai berikut.

```
// Penggalan kode 4
01 void javaFinally () {
02   resources = new resources();
03   try {
04     // throw eksepsi
05   }
06   finally {
07     resources.cleanup;
```

```
08    }
09 }
```

C++ tidak mempunyai fitur yang secara langsung setara dengan blok *finally*. C++ menggunakan *smart pointer* untuk melakukan pembersihan obyek jika blok telah ditinggalkan. Ide dasar dari *smart pointer* adalah implementasi suatu klas yang kompatibel dengan *pointer* standar dan mempunyai destruktor yang melakukan pembersihan. STL mendefinisikan klas *auto_ptr<>* untuk mengimplementasikan *pointer* tipe ini. Implementasi tipe *pointer* ini disebut teknik *destructive copy*. Ini berarti hanya akan ada satu pointer *auto_ptr<>* yang mempunyai obyek untuk ditunjuk. Jika *auto_ptr<>* statusnya menjadi *deleted*, destruktor akan melakukan *delete* di obyek yang ditunjuk. Ini manjamin bahwa destruksi *pointer* ini juga akan menghapus obyek yang ditunjuknya.
Penggalan kode 5 berikut ekuivalen dengan kode 2 menggunakan *auto_ptr<>* yang definisinya berada di processor *directive #include <memory>*. Perbedaan utamanya adalah *pointer* integer (int *) diganti dengan *std::auto_ptr<int>*.

```
// Penggalan kode 5
01 #include <memory>
02 void cppAutoPtr () {
03    std:: auto_ptr<int> x(new int (0));
04    std::auto_ptr<int> z;
05    {
06      *x = 3;
07      std:: auto_ptr<int> temp (new int (*x));
08      int& y = *temp;
09      y++;
10      z = temp;
11    }
12    (*x) ++;
13    ++(*z);
14 }
```

Perbedaan lain adalah penanganan referensi y. Obyek yang sinonim dengan y sekarang juga dikelola oleh *auto_ptr<>* yang kita beri nama *temp*. Untuk mengirim obyek ini keluar dari blok dalam kita harus mengkopi *pointer* ke peubah yang tersedia di blok luar. Hal ini dilakukan di baris 10. Sekarang peubah z yang bertanggung jawab pada kepemilikan obyek ini. Ia hanya akan dihapus di baris 14 bukannya baris 11. Perlu di catat bahwa *auto_ptr<>* bukan *pointer* generik meskipun ia memiliki sejumlah perilaku yang tidak biasa seperti *transfer of ownership*, *destructive copy* dan sebagainya (Josuttis, 1999). Tipe *pointer* lain yang dapat diimplementasikan dengan cara yang sama adalah *garbage collected pointer*. Berbeda dari implementasi *transfer ownership* yang dilakukan oleh *auto_ptr<>* kita hanya perlu mengenalkan *counter* . *Counter* ini akan naik satu nilainya *pointer* lain menunjuk obyek yang sama dan akan turun satu nilainya jika salah satu *pointer* yang merujuk obyek ini mengambil obyek yang telah dihapus atau obyek baru yang di-*assign* ke *pointer* lain. Jika nilainya sama dengan 0, ia akan menghapus obyek yang ditunjuknya. Meskipun STL tidak memuat *pointer* ini tapi sejumlah aplikasi lain dari *pointer* ini dapat ditemui di Josuttis (1999) dan Alexandrescu (2001). Secara singkat destruktor C++ dapat digunakan secara fleksibel seperti halnya kita menggunakan blok *finally* nya Java.

Akses ke Basis Data

Perbandingan terakhir yang dilakukan adalah implementasi kedua bahasa untuk mencari solusi suatu aplikasi basis data. Fungsionalitas dasar ke basis data dipenuhi oleh dua metode yaitu metode *get()* dan metode *release()*.

Penggalan kode 6 adalah implementasi langsung bahasa Java dari teknik *finally* yang disajikan di penggalan kode 4. Di baris 2, basis data di akses dari *pool*-nya, tubuh dari blok *try* melakukan operasi padanya dan akhirnya blok *finally* mengembalikan basis data ke *pool*-nya.

```
// Penggalan kode 6
01 void dbPool ( pool myPool) {
02   handle h = myPool.get();
03   try {
04     // throw exception here
05   }
06   finally {
06     myPool.release(h);
07   }
08 }
```

Penggalan kode C++ berikut adalah implementasi langsung dari penggalan kode 5 sebelumnya. Baris pertama mendeklarasikan klas *template* dengan dua tipe parameter R (*resources*) dan P (*pool*). Deklarasi ini dibutuhkan agar klas bisa menggunakan kembali sumber-sumber lain di tubuh klas *gc_resources*.

```
// Penggalan kode 7
     01 template <class P, class R>
     02 class gc_resources {
     03   public:
     04     gc_resources (P a_pool):
     05      _pool (a_pool),
     06      _resources (_pool.get()) {
     07     }
     08
     09     ~gc_resources () {
     10       _pool.release (_resources);
     11    }
     12
     13    R* operator ->() {
     14       return _resources;
     15    }
     16
     17   private:
     18     P& _pool;
     19     R * _resources:
     20 };
```

Konstruktor di baris 04 membutuhkan *pool* sebagai argumen, disimpan secara lokal dan melakukan operasi *get()* padanya. Hasilnya, *handler* atau *pointer* ke *handler*, juga disimpan secara lokal. Destruktor di baris 09 mengembalikan basis data yang diambilnya ke *pool* menggunakan metode *release()*. Fungsi di baris 13 membebalebihi operator *dereference* -> untuk mengakses *handler* dari luar klas *gc_resources*.

Secara sepintas kedua implementasi ekuivalen. Keuntungan dari implementasi C++ adalah kita tidak perlu menyimpan sejumlah baris saat kita mempunyai basis data. Ini artinya tidak ada tugas yang dilupakan. Dengan kata lain pembersihan kode otomatis terpasang. Di sisi lain, implementasi Java tidak membutuhkan klas khusus, dalam arti kita tidak perlu tahu bagaimana merancang klas tersebut. Implementasi Java lebih langsung dan mudah untuk dipahami. Kekurangan kode Java adalah

pemrogram harus melakukan kewajibannya dengan benar untuk menghindari terjadinya duplikasi yang bisa mengarah ke *error prone*.

PENUTUP

Tidak mudah untuk menyimpulkan mana yang lebih baik antara C++ dan Java. Untuk pengelolaan memori, kedua bahasa menggunakan konsep yang jauh berbeda. Java menggunakan GC secara otomatis, sedangkan C++ menggunakan konstruktor, destruktor dan *smart pointer*. Meskipun C++ baku mengijinkan penggunaan GC, tetapi secara praktis GC tidak digunakan. Untuk pembersihan data C++ menggunakan destruktor sedangkan Java menggunakan *finalizer* yang sangat baik untuk membersihkan memori tetapi tidak untuk sumber-sumber lain sehingga ada klaim *finalizer* Java tidak *reliable*. Untuk menangani eksepsi kedua bahasa menggunakan konstruksi *try-catch*. Meskipun tidak mempunyai blok *finally*, destruktor C++ dapat digunakan secara luwes menggunakan smart *pointer* untuk menyamai fungsi blok *finally*-nya Java. Untuk akses ke basis data sangat tergantung pada ukuran proyek. Untuk proyek skala kecil Java lebih menguntungkan sedangkan untuk skala besar C++ jauh lebih menarik karena memungkinkan kita untuk mengeliminasi duplikasi kode dengan cara membentuk klas kecil.

DAFTAR PUSTAKA

Gosling, J., Joy, B., Steele, G. and Bracha, G. (2000). *The Java Language Specifications* (2nd ed). New Jersey: Addison Wesley.

Josuttis, N. M. (1999). *The C++ Standard Library*. New Jersey: Addison Wesley.

Kapoor, M. (2003). *Using the C/C++ Garbage Collection Library libgc*. SunMicro System.